# Inter-Context Control-Flow and Data-Flow Test Adequacy Criteria for nesC Applications[*]

Zhifeng Lai and S.C. Cheung[†]
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Kowloon, Hong Kong
{zflai, scc}@cse.ust.hk

W.K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

## ABSTRACT

NesC is a programming language for applications that run on top of networked sensor nodes. Such an application mainly uses an interrupt to trigger a sequence of operations, known as contexts, to perform its actions. However, a high degree of inter-context interleaving in an application can cause it to be error-prone. For instance, a context may mistakenly alter another context's data kept at a shared variable. Existing concurrency testing techniques target testing programs written in general-purpose programming languages, where a small scale of inter-context interleaving between program executions may make these techniques inapplicable. We observe that nesC blocks new context interleaving when handling interrupts, and this feature significantly restricts the scale of inter-context interleaving that may occur in a nesC application. This paper models how operations on different contexts may interleave as inter-context flow graphs. Based on these graphs, it proposes two test adequacy criteria, one on inter-context data-flows and another on inter-context control-flows. It evaluates the proposal by a real-life open-source nesC application. The empirical results show that the new criteria detect significantly more failures than their conventional counterparts.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*.

## General Terms

Reliability, Experimentation, Verification.

## Keywords

Test Adequacy Criteria, Software Testing, Networked Embedded System.

* This research was partially supported by the Research Grants Council of Hong Kong under grant numbers 612306, 111107, and HKBU 1/05C, National Science Foundation of China under grant number 60736015, National Basic Research of China under 973 grant number 2006CB303000, and a grant of City University of Hong Kong under grant number CityU 7002324.

† Correspondence author.

## 1. INTRODUCTION

NesC language [10] is designed for programming wireless sensor network (WSN) applications that are deployed on a collection of small low-powered low-capability devices known as motes. Each mote usually has sensing and wireless communication capabilities. WSN applications are useful for monitoring their physical environments. Most nesC applications running on motes are interrupt-driven, so hardware processors of these motes can *sleep* and conserve energy when there is no interrupt.

In general, event-driven programming is popular for handling interrupts in networked embedded systems running on top of TinyOS [14], TinyGALS [6], or SOS [12]. In the event-driven programming paradigm, developers implement application logic through cooperative event handlers. A high degree of interrupt-based concurrency complicates the program design to carry out the desired cooperation [10]. For instance, an abrupt interrupt may occur during the time when an event is being handled. The occurrence of the interrupt may potentially interfere with the event handler through unforeseen dependencies between the event handler and the code that handles the interrupt.

Programming nesC applications is an error-prone task: developers of nesC applications need to explicitly handle numerous interrupts in their code. Researchers have reported the problems of data races [10][13], interface-contract violation [4], and memory safety [7] in these applications. Although nesC is designed to ease static analysis, existing static checking [13] and runtime verification [4][7] techniques do not provide a satisfactory solution to these problems. For instance, efficient race detectors [10] may not identify some race conditions (c.f. TinyOS-Help[1], a major discussion mailing-list of TinyOS related problems); powerful race detector [13], however, is subject to scalability and precision problems. Even if the errors identified by these techniques have been corrected, the modified nesC applications may still have latent faults.

Testing is a major means to assure the quality of software. However, existing testing techniques for sequential programs [9][11][16] address the interactions among interrupts inadequately. In sequential programs, subroutines call one another (termed a *calling context*). In nesC applications, an interrupt from the TinyOS initiates a new calling context. Because multiple interrupts may occur in the same period, an execution of a nesC application may have concurrent calling contexts. These calling contexts may interleave and share resources, such as CPU and memory, so that the application can use these resources economically because of tight resource constraints in sensor motes.

[1] http://www.mail-archive.com/tinyos-help@millennium.berkeley.edu/msg13570.html

Existing testing techniques for concurrent programs [20][33] rely on the identification of synchronization primitives to analyze interleaving among concurrent subroutines. However, the number of possible interleaving for applications written in general-purpose languages (e.g., Java) is often huge. Therefore, adequate test coverage of interleaving is usually difficult. This impedes the utility of these techniques in practice.

However, with proper adaptation, these concurrency testing techniques can be made more tractable for nesC applications because of the simple scheduling policy adopted by TinyOS. A typical nesC application consists of two major kinds of subroutines: namely tasks and interrupt handlers. The execution of a subroutine should be short in duration to allow the application to respond to the next interrupt. Thus, a subroutine often posts follow-up subroutines to compute the remaining job [10]. As we explain in this paper, we observe three scheduling heuristics of nesC applications to make testing more effective. (1) An interrupt handler can preempt any subroutine being executed. (2) Execution of interrupt handlers is *non-blocking*: they run to completion unless they are preempted by other interrupt handlers. (3) A newly posted task will not be executed until all the outstanding ones have finished. Suppose *A* and *B* are two arbitrary nesC subroutines. Consider the scenario of interleaving between *A* and *B*. At the time the subroutine *A* finishes, on one hand, the subroutine *B* blocks the execution of follow-up tasks, no matter it is posted by *A* or *B*, until *B* also finishes because of the second and third heuristics. On the other hand, the execution of *B* may be preempted by an interrupt handler because of the first heuristics. This effectively restricts the degree of interleaving that can be occurred in nesC applications. We observe that these three heuristics collectively make the interleaving between subroutines more tractable than the interleaving between two arbitrary threads in, for instance, Java programs. Intuitively, this improvement in tractability saves efforts in analyzing source code such as conducting reachability analysis and reduces the effort in testing whether the interactions between subroutines may cause failures.

In this paper, we propose a framework to test nesC applications based on these observations. We firstly model the potential execution orders of tasks in a nesC application as *task graphs*. Based on task graphs, we propose *Inter-Context Flow Graphs* (ICFGs) to model the behaviors of nesC applications for testing purposes. ICFG captures not only control transfers in individual subroutines, but also the interactions among subroutines in concurrent calling contexts initiated by interrupts. We further propose control-flow and data-flow test adequacy criteria based on ICFGs to measure the coverage of test suites for testing the interactions among subroutines. We evaluate our proposed testing criteria using an open-source structural health monitoring application [19]. Experimental results show that our criteria are on average 21 percent more effective in exposing faults than their conventional counterparts.

This paper makes the following main contributions:

(1) We develop a notation, ICFG, to model nesC applications for testing purpose.
(2) We formulate two new test adequacy criteria for measuring the quality of test suites to test nesC applications.
(3) We report an experiment that compares the effectiveness of our two new criteria with that of two existing representative criteria: all-branches and all-uses.

We organize the rest of the paper as follows. Section 2 outlines the fundamentals of TinyOS and nesC relevant to this paper. Section 3 uses an example to motivate our work. Section 4 presents our approach to model nesC applications and to test them. Section 5 presents an experimental evaluation using an open-source project. Section 6 reviews related work and Section 7 summarizes this paper.

## 2. FUNDAMENTALS

Let us review TinyOS's execution model [14] as well as the nesC language constructs [10] related to our modelling approach.

NesC applications are mainly driven by interrupts. When an interrupt occurs, TinyOS translates it as an event and propagates that event to a target nesC application. An application subroutine running asynchronously in response to an interrupt event is tagged by the keyword *async* (stands for *asynchronous*). For convenience, we call such a subroutine an asynchronous event handler (AEH). Event handlers that are not tagged by the "async" keyword are only reachable from tasks [10]. It is a common development idiom to make the execution of an AEH short in duration to ensure overall application responsiveness [34]. Developers usually design an AEH to partly address the interrupt and post *tasks* to complete the rest of the action. Every AEH can preempt any other program execution unless the later is executing statements in an *atomic* block, which nesC guarantees to execute the block atomically. However, unlike the conventional call-and-return semantics, posting a task does not mean the task is executed immediately [28]. Instead, the posting mechanism simply puts the task in a task queue as the last element and returns immediately. It defers the execution of the task until there are no more outstanding interrupts or tasks before it in the queue. Once scheduled, a task runs to completion with respect to other tasks.

A ***calling context***, or simply ***context***, is a sequence of subroutines where contiguous subroutines have call relations [2]. In nesC, the root calling contexts can be synchronous or asynchronous. The synchronous root calling context is the `main` routine similar to that in a C++ program; the asynchronous root calling contexts are interrupt handlers. These interrupt handlers are not reachable from the `main` routine (in the sense of call graphs in static analysis) and multiple calling contexts may interleave among themselves.

## 3. A MOTIVATING EXAMPLE

Figure 1 gives a module called OscilloscopeM (denoted by *OM*), adapted from the TinyOS distribution[2]. We adapt the code for ease of explaining a nesC fault. In nesC, any module implementing application logic is a component. In our example, OM declares two shared variables (`packetReadingNumber` and `msg`) at line #10 and #12, defines four tasks (`task1`, `task2`, `task3`, and `task4`) in between lines #13 and #38, and implements three event handlers (`PhotoSensor.dataReady`, `SendMsg.sendDone`, and `TempSensor.dataReady`) in between lines #39 and #59. OM requests the underlying hardware platform to sample readings of photo and temperature sensor, keeps the samples in a buffer, and sends the readings when certain conditions meet. OM could be used for controlling indoor housing conditions.

Let us consider a typical scenario of OM. When a photo sensing datum is available, TinyOS stores the datum into a shared buffer by invoking the AEH `PhotoSensor.dataReady` (#39~#51). The shared buffer is a field of the variable `msg` (#12) in the TinyOS packet format `TOS_Msg`. OM accesses the buffer by the pointer

[2] http://tinyos.cvs.sourceforge.net/tinyos/tinyos-1.x/apps/Oscilloscope/

`pack` (#42). When the size of the buffer reaches a predefined threshold value `(BUFFER_SIZE)`, this AEH posts an instance of `task2` to process the buffer.

```
#1: module OscilloscopeM {
#2:    provides interface StdControl;
#3:    uses {
#4:       interface Leds;
#5:       interface ADC as PhotoSensor;
#6:       interface ADC as TempSensor;
#7:       interface SendMsg; /* other interfaces */
#8:    }
#9: } implementation {
#10:   uint8_t packetReadingNumber;                    // shared variable
#11:   // uint8_t lock = 0;
#12:   TOS_Msg msg;                                    // shared variable
#13:   task void task4() { /* do something */ }
#14:   task void task3() { /* do something */ }
#15:   task void task2() {
#16:      OscopeMsg *pack;
#17:      static uint32_t i;
#18:      uint32_t start = i;
#19:      atomic {                                     // atomic block
#20:         pack = (OscopeMsg *)msg.data;
#21:         packetReadingNumber = 0;
#22:         for (; i < start + 100000 && i < 200001; i++) { /* process sensing data */ }
#23:      }
#24:      if (i < 200001) {
#25:         post task2(); // post another instance to process remaining data
#26:         return;
#27:      } // else (finish processing the data): send the packet
#28:      i = 0;
#29:      pack->channel = 1;
#30:      pack->sourceMoteID = TOS_LOCAL_ADDRESS;
#31:      if (call SendMsg.send(TOS_UART_ADDR, sizeof(OscopeMsg), &msg)) {
#32:         call Leds.yellowToggle();
#33:      }
#34:   }
#35:   task void task1() { /* do something */
#36:      post task3();
#37:      post task4();
#38:   }
#39:   async event result_t PhotoSensor.dataReady(uint16_t data) {
#40:      OscopeMsg *pack;
#41:      // atomic { if (lock == 1) return; }
#42:      pack = (OscopeMsg *)msg.data;
#43:      atomic {
#44:         pack->data[packetReadingNumber++] = data;
#45:      }
#46:      if (packetReadingNumber == BUFFER_SIZE) {
#47:         // lock = 1;
#48:         post task2();
#49:      }
#50:      return SUCCESS;
#51:   }
#52:   event result_t SendMsg.sendDone(TOS_MsgPtr sent, result_t success) {
#53:      // atomic lock = 0; /* notify the availability of the message buffer */
#54:      return SUCCESS;
#55:   }
#56:   async event result_t TempSensor.dataReady(uint16_t data) { // do something
#57:      post task1();
#58:      return SUCCESS;
#59:   } /* other commands, events, and tasks */
#60: }
```

**Figure 1. A component using a split-phase interface SendMsg.**

Buffer processing often needs intensive computation (#22). Owning to the simple scheduling policy adopted by TinyOS, performing the entire computation in one single task may starve the executions of the other tasks. Therefore, it is a general advice for developers to split an intensive computation into multiple tasks [28]. In OM, the buffer processing is carried out collectively by multiple instances of `task2`. Each instance processes a fraction of the buffer and then posts a follow-up instance of `task2` to handle the rest (#25). A follow-up instance of `task2` will not execute until the immediately preceding instance has completed. The final `task2` instance further invokes `SendMsg.send` (#31) to send out a packet (`msg`) that stores the entire processed result of the buffer to a base station. The call returns immediately. The underlying communication layer returns `SUCCESS` if it can send the packet. This scenario is depicted in Figure 2, where `BUFFER_SIZE` is 2.

The OM code contains at least one fault. An AEH `dataReady` (#39) may occur and fill the buffer with new data during an arbitrary instance of `task2`, which is processing the data buffer. For instance, suppose that a processed result is stored in the first slot of the buffer. If `dataReady` preempts `task2` at #29 before `task2` sends out the packet in full, a new sensing datum may overwrite the result kept in that slot. `Task2` may thus send an inappropriate packet after it resumes its execution. A correct implementation should suppress all `dataReady` invocations during the interval, say, between $t_3$ and $t_{10}$ in Figure 2. A solution is to add a locking mechanism to the component. Developers can achieve this by uncommenting the underlined statements in Figure 1.

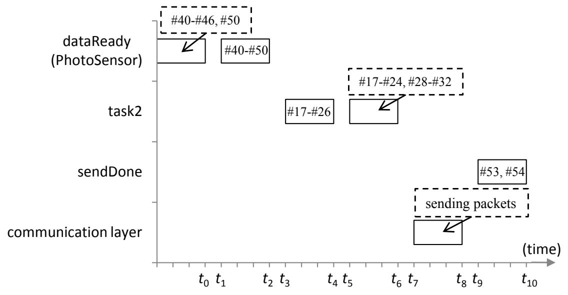


**Figure 2. Timeline diagram of a normal scenario.**

Existing testing techniques that do not capture detailed concurrent execution semantics of nesC applications may not easily expose this fault. Conventional intra- and inter-procedural control-flow and data-flow testing techniques [9][11][16] are developed for sequential programs, and thus neglect the concurrency caused by interrupts. For example, a test suite enforcing the execution of the above scenario satisfies the all-uses criterion [9]. Still, as we have shown, such a test suite does not effectively reveal the failure. The granularity similar to that of the event-driven approach to GUI testing [24] is not fine enough to reveal a failure related to this fault. For example, individual executions of event handlers without any preemption satisfy the all-events criterion [24]. Nevertheless, this failure can only be revealed in the presence of preemption. To expose the fault, a test suite is desirable to exercise certain preemption scenarios. For example, a test suite enforces `dataReady` (#39) to preempt `task2` during the interval from $t_3$ to $t_6$ can reveal a failure if new sensing datum is different from the computation result. Application posting graphs [25] do not capture resumption of tasks after they have been preempted. Therefore, adequacy criteria developed based on this representation may be inadequate. For example, the scenario, where `dataReady` (#39) preempts `task2` at #29 during $t_5$ to $t_6$, exposes the fault. However, the application posting graph of OM is unable to show the definition-clear [1] sub-path (#44→#46→#50→#29→#30→#31) for the variable `msg`. In summary, testing techniques that cover preemption (e.g., #28→#39) and resumption (e.g., #50→#29) scenarios may increase the chances to expose this fault.

On the other hand, applying existing concurrency testing techniques [20][33] that do not leverage the scheduling heuristics in TinyOS could be costly. Assuming that developers intend to test the interaction between `dataReady` (#39) and `task2`, and these two subroutines are executed in two different contexts in

Java. The number of blocks (defined in Section 4.1) in `dataReady` and `task2` are 5 and 11, respectively (these figures exclude entry and exit blocks; note the return statements at #26 and #50 are associated with exit blocks). Therefore, in conventional concurrent systems, the number of possible interleaving between these two subroutines is (5+11)!/(5! • 11!), which is 4368. On the other hand, the corresponding number of possible interleaving in nesC is only 11, which is equivalent to the number of scenarios that `dataReady` preempts each block of `task2`. We believe that it is less smart to analyze source code based on these 4368 scenarios and to generate tests to cover all of them.

## 4. REPRESENTATION

In this section, we introduce *Inter-Context Flow Graph* (ICFG) to model a nesC application.

### 4.1 Preliminaries

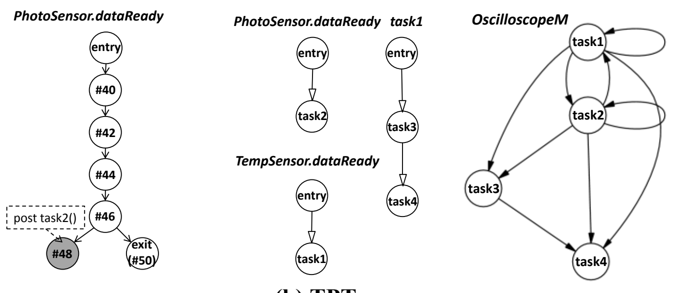
A *block* is a collection of statements that are executed without being preempted. Syntactically, a block is either (1) a statement (except for a return statement) outside any atomic block, or (2) a group of statements in an atomic block. NesC guarantees that statements in an atomic block are non-preemptable [10]. There are two special blocks for each subroutine. An *entry* block tags the entrance of a subroutine. An *exit* block collectively represents implicit and explicit return statements in a subroutine. A *Control-Flow Graph* (CFG) of a subroutine is a directed graph where each node represents a statement and each edge represents the flow of control between statements. A *Flow Graph* (FG) is similar to a CFG except that nodes represent blocks and edges represent the flow of control between blocks.

Conventional data-flow analysis for a subroutine is often based on CFGs [9]. Variable occurrences in a CFG are classified as either *definitions*, at which values are stored, or *uses*, at which values are fetched. A *def-use association* (*dua*) is a triple ($v$, $d$, $u$) such that there is a *definition-clear* sub-path from $d$ to $u$ with respect to $v$, where $v$ is a variable, $d$ is a node at which $v$ is defined, $u$ is a node or an edge at which $v$ is used. A sub-path is definition-clear with respect to $v$ if no node in the path redefines $v$. We say a test case $t$ *covers* a *dua* ($v$, $d$, $u$) if $t$ causes the program to execute a sub-path that goes from $d$ to $u$ and the sub-path is definition-clear with respect to $v$.

To leverage nesC's scheduling heuristics, we augment the CFGs with annotations about task preemptions and task postings. We build a *task posting tree* to capture the relative ordering of task post statements in each subroutine, and a *task graphs* to express the relative execution orders of tasks in each component. Task preemptions are directly expressed in ICFG defined in next section. Formal definitions of task posting trees and task graphs are given as follows.

DEFINITION 1. *A* **Task Posting Tree (TPT)** *for a subroutine u is a directed tree of 2-tuple $G_{TPT}(u) = (N_u, E_u)$, where $N_u$ is a set of nodes and $E_u$ is a set of directed edges. Each node $n_i \in N_u$ is a post statement in $G_{CFG}(u)$. An edge $<n_i, n_j>$ in $E_u$ has the following two properties: (1) $n_i$ dominates $n_j$ in $G_{CFG}(u)$; (2) if another post statement $n_k \in N_u$ ($n_k \neq n_i$) dominates $n_i$, $n_k$ also dominates $n_j$ in $G_{CFG}(u)$.*

A node $n_i$ *dominates* a node $n_j$ if and only if every directed path from the entry node to $n_j$ (not including $n_j$) contains $n_i$. A *dominator tree* [1] of $u$ is a tree in which the root node is the entry node, and each node dominates only its descendants in the tree. We build a $G_{TPT}(u)$ based on the dominator tree of $u$'s CFG, by removing every node which does not model a *post* statement in $u$'s dominator tree. If such a node is a leaf node in the dominator tree we directly remove it. If such a node is an interior node, we remove it and connect its unique parent with all its children. An example of dominator tree for the AEH, `PhotoSensor.dataReady`, is shown in Figure 3(*a*). Intuitively, if a post statement "`post task1`" is an ancestor of another post statement "`post task2`" in $u$'s dominator tree, `task1` will be posted before `task2` in any of $u$'s executions. We use the algorithm presented in [1] to compute $u$'s dominator tree from $G_{CFG}(u)$. Figure 3(*b*) shows a collection of TPTs for the OM example. Nodes in Figure 3(*b*) model post statements, but we omit the *post* prefixes to ease the subsequent discussion. For example, we refer to a node representing "`post task1`" as "`task1`" in Figure 3(*b*).



(*a*) **Dominator tree.** **(b) TPTs.** (*c*) **TG.**

**Figure 3. Dominator tree, TPTs and TG.**

DEFINITION 2. *A* **Task Graph (TG)** *for a component C is a directed graph of 2-tuple $G_{TG}(C) = (N_C, E_C)$, where $N_C$ is a set of tasks in C, and $E_C$ is a set of directed edges representing execution orders among these tasks. There is an edge $<n_i, n_j>$ in $E_C$ if task $n_j$ is possible to be executed after $n_i$ completes its execution.*

A component $C$ may define multiple tasks. The task graph of $C$, $G_{TG}(C)$, represents potential execution orders of tasks in $C$. We build a $G_{TG}(C)$ based on TPTs of $C$ by the algorithm shown in Table 1.

**Table 1. Algorithm to compute a task graph in a component.**

```
Algorithm: getTaskGraph
Input:
    $G_{TPT}(u)$: $u$ is a subroutine of a component $C$
Output:
    $G_{TP}(C) = (N_C, E_C)$: a task graph of $C$
      $N_C = \{T \mid T$ is a task in $C\}$; $E_C = \{\}$.
      for each subroutine $u$
         for each post statement $T_i$ in $G_{TPT}(u)$
            for each $T_i$'s descendent $T_j$
               add $(T_i, T_j)$ to $E_C$;
      for each $T_i$ in $N_C$
         for each post statement $T_j$ in $G_{TPT}(T_i)$
            add $(T_i, T_j)$ to $E_C$;
      for each AEH $a$
         for each post statement $T_i$ in $G_{TPT}(a)$
            for each $T_j$ in $N_C$
               add $(T_i, T_j)$ to $E_C$;
```

Let us explain the algorithm. In the first outmost loop, an edge $<T_i, T_j>$ of a $G_{TPT}(u)$ means that $T_i$ is posted before $T_j$ in any execution of $u$. For instance, this step adds the edge `task3`→`task4` to $E_C$. In the second outmost loop, if $T_j$ is posted from a task $T_i$, $T_j$ may then execute right after $T_i$ has completed its execution, if there is no outstanding interrupt and the task queue is empty when $T_j$ is posted. For instance, this step adds the edge `task1`→`task3` to $E_C$. In the third outmost loop, if $T_i$ is posted from an AEH $a$, posting of $T_i$ may happen before any task posting action in $C$. Therefore, $T_i$ may execute before any task defined in $C$. For instance, this step adds edges from `task2` to all other tasks to $E_C$. Figure 3($c$) shows the task graph for OM.

TGs only express interactions among tasks. In the next section, we introduce ICFGs to express interactions among interrupt handling subroutines and tasks.

## 4.2 Inter-Context Flow Graph

As we have described in Section 1 and Section 2, an interrupt handler may preempt an executing subroutine, and then let the preempted subroutine resume its execution. When the subroutine suspends, the interrupt handler may change the content of shared storages kept by a previously suspended subroutine. The suspended subroutine or a newly posted task may use these storages with altered contexts subsequently. It may produce undesirable outcomes and cause a program failure. Our Inter-Context Flow Graph captures such preemption and resumption of subroutine executions as edges among task graphs.

An Inter-Context Flow Graph (ICFG) of a component $C$, $G_{ICFG}(C)$, is defined based on $G_{TG}(C)$. $G_{ICFG}(C)$ aims at relating the flow graphs of individual subroutines in $C$ by control-flow edges caused by interrupts and deferred execution of tasks. A sample ICFG of the OM example is shown in Figure 4. We hide some blocks in the flow graphs for the clarity of presentation. The idle node ($n_{idle}$) models the status of the application when an application has no outstanding subroutines to be executed. In other words, the application idles and waits for incoming interrupts. All components share the same node $n_{idle}$ ($n_0$ in Figure 4). Besides the control-flow edges in individual subroutines' FGs, as we have mentioned above, an ICFG of a component adds a set of edges to link up these FGs.

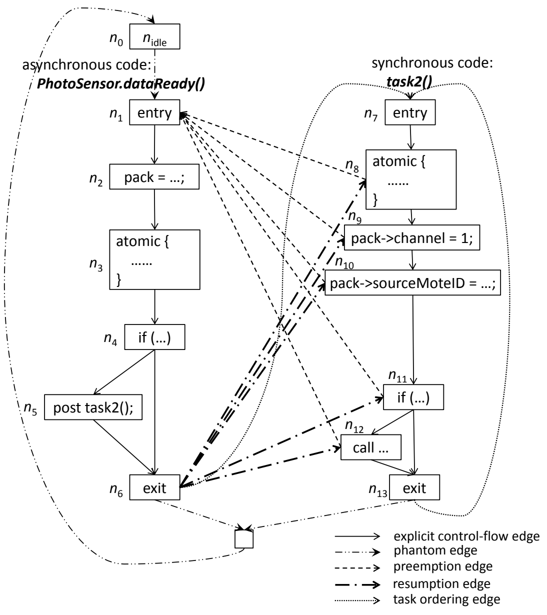


**Figure 4. A partial ICFG for OM.**

We have identified four types of control-flow edges, each serving a unique purpose according to the programming model of nesC. As we have explained, an invocation of an interrupt handler will result in a new calling context. A phantom edge thus serves to link FGs of interrupt handlers and tasks in different contexts. An interrupt handler may suspend an executing subroutine, and the preempted subroutine may resume its execution after the interrupt handler finishes. A preemption edge models a suspension of a calling context when an interrupt occurs, while a resumption edge models the resumption of a calling context when an interrupt handler finishes. A task ordering edge models how two task postings define the execution order of the associated posted tasks. Such an edge captures the interleaving of posted tasks in nesC applications.

We incorporate these kinds of edges into our model to facilitate us to conduct inter-context control-flow and data-flow analysis. For instance, the shared buffer of OM in Figure 4 is defined at node $n_3$. Furthermore, the buffer can be used by `task2` at node $n_{10}$ through the sub-path $n_3 \rightarrow n_4 \rightarrow n_6 \rightarrow n_9 \rightarrow n_{10} \rightarrow n_{11}$. This sub-path traverses the resumption edge $n_6 \rightarrow n_9$. Readers can interpret the use of the other newly formulated edges similarly. Below, we present how to build these edges.

**Phantom edges**. There are two sub-types.

(1) For each AEH $a$, add an edge from $n_{idle}$ to the entry block of $G_{FG}(a)$. It models an invocation of an interrupt hander when the program is at the idle state. An example is $n_0 \rightarrow n_1$ in Figure 4.

(2) For each subroutine $u$, add an edge from the exit block of $G_{FG}(u)$ to $n_{idle}$. It models the transfer of control from a subroutine to the idle state when the subroutine has completed its operations and there is no other outstanding subroutines to process. The edges $n_6 \rightarrow n_0$ and $n_{13} \rightarrow n_0$ in Figure 4 are examples.

**Preemption edges**. For each block $B$ in every $G_{FG}(u)$ (except the entry and exit blocks) of $C$, and for each AEH $a$, add an edge from $B$ to the entry block of $G_{FG}(a)$.

A preemption edge from $B$ to the entry block of $G_{FG}(a)$ represents that $a$ can preempt $B$ before $B$'s execution. For instance, $n_9 \rightarrow n_1$ in Figure 4 is a preemption edge.

**Resumption edges**. For each AEH $a$ and block $B$ in every $G_{FG}(u)$ (except the entry and exit blocks) of $C$, add an edge from the exit block of $G_{FG}(a)$ to $B$.

A resumption edge from the exit block of $G_{FG}(a)$ to $B$ denotes that when $a$ preempts $B$, $B$ can resume its execution after $a$ has completed its execution. For instance, $n_6 \rightarrow n_9$ in Figure 4 is a resumption edge.

**Task ordering edges**. There are two sub-types.

(1) For each AEH $a$, and for each task $T_i$ in $G_{TPT}(a)$, add an edge from the exit block of $G_{FG}(a)$ to the entry block of $G_{FG}(T_i)$. It expresses that if an AEH $a$ posts a task $T$, $T$ can be executed after $a$ has completed its execution. For instance, $n_6 \rightarrow n_7$ in Figure 4 is an example.

(2) For each edge $\langle T_i, T_j \rangle$ in a component $C$'s task graph $G_{TG}(C)$, add an edge from the exit block of $G_{FG}(T_i)$ to the entry block of $G_{FG}(T_j)$. The execution order of $T_i$ and $T_j$ represented by this type of task order edge is defined by the notion of task graph. For instance, $n_{13} \rightarrow n_7$ in Figure 4 is an example.

DEFINITION 3. *An* **Inter-Context Flow Graph (ICFG)** *for a component C is a directed graph of 6-tuple* $G_{ICFG}(C) = (G_{FG}, n_{idle}, E_{phan}, E_{preem}, E_{resum}, E_{task})$*, where* $G_{FG}$ *are FGs of all subroutines in C;* $n_{idle}$ *is an idle node;* $E_{phan}$*,* $E_{preem}$*,* $E_{resum}$*, and* $E_{task}$ *are sets of phantom, preemption, resumption, and task ordering edges.*

Let us use a scenario to illustrate the ICFG in Figure 4. The application starts at $n_0$, waiting for incoming interrupts. When the interrupt `PhotoSensor.dataReady` occurs, the control flow transfers from $n_0$ to $n_1$. Suppose that there is no interrupt preempting this handler and it executes a post statement to post an instance of `task2` ($n_5$). The task queue enqueues the `task2` instance. This instance is executed only after the handler completes its execution modeled by <$n_6$, $n_7$>. Suppose another instance of interrupt `PhotoSensor.dataReady` occurs right before $n_9$ is about to be executed. The interrupt handler has preempts $n_9$ and is modeled by the preemption edge <$n_9$, $n_1$>. The handler posts another instance of `task2` ($n_5$), yet the newly posted instance of `task2` will not be executed immediately after the handler completes its execution. Rather, the first instance of `task2` resumes its execution as modeled by the resumption edge <$n_6$, $n_9$>, and the execution of the second `task2` instance will traverse the edge <$n_{13}$, $n_7$> after the first instance of `task2` has completed its execution. If no more interrupts occur after both instances complete their executions, the control flow will transfer from $n_{13}$ to $n_0$, which models the software sleeps and waits for more interrupts to occur.

## 4.3 Test Adequacy Criteria

In this section, we propose our test adequacy criteria to measure the quality of test suites for nesC applications. Test engineers use test adequacy criteria as *stopping rules* to decide when testing has achieved objectives or as *guidelines* for selecting extra test cases [35]. We have discussed that there are control flows among interrupt handling subroutines and tasks, and we have formulated them in ICFGs. We have further discussed in the motivating example that using these edges, faults relevant to nesC applications can be effectively exposed. We thus propose using control-flow (all-inter-context-edges) and data-flow (all-inter-context-uses) criteria to serve this purpose. Intuitively, our all-inter-context-edges criterion extends the notion of all-branches criterion [16] and the all-inter-context-uses criterion extends the notion of all-uses criterion [9].

CRITERION 1. *A test suite* T *satisfies the* **all-inter-context-edges** *adequacy criterion for a nesC application if and only if for each edge* E *in every component's ICFG, at least one test case in* T *exercises* E.

We say an adequacy criterion $C_1$ subsume an adequacy criterion $C_2$, if every test suite that satisfies $C_1$ also satisfies $C_2$ [35]. The all-inter-context-edges criterion subsumes the all-branches criterion because all-inter-context-edges covers control-flow edges in individual FGs of the application as well as the new kinds of edges captured by ICFGs.

**Table 2. Algorithm to compute def-use associations of shared variables in a component.**

```
Algorithm: getDUs
Input:
    G_ICFG(C): an ICFG for a component C
    V_S^C: a set of shard variables in C
Output:
    DU: a set of def-use associations for shared variables in component C

    DU = {};
    for each block B in G_ICFG(C)
        UPEXP(B): the set of upwards exposed uses in block B;
        for each use u in UPEXP(B)
            RD(B): the set of reaching definitions in block B;
            for each definition d in RD(B)
                if ((d is a definition of v) && (u is a use of v) && (v is in V_S^C))
                    add (v, d, u) to DU;
```

Data-flow adequacy criteria [9][11][20][33] are remarkable for selecting test cases to expose faults related to improper data uses. Data-flow criteria require that test cases cause the application to exercise paths from locations that define variables to locations that subsequently use these variables. Our ICFGs capture control flows across concurrent calling contexts and therefore can identify data-flow associations more precisely. Table 2 shows an algorithm that computes the def-use associations of shared variables in components based on ICFGs. An upwards exposed use of block $B$ is a use of variable $v$ in $B$ such that no prior definition of $v$ occurs in $B$. For a block $B$, the set UPEXP($B$) can be computed locally in that block. A definition $d$ with respect to a variable $v$ *reaches* a point $p$ if there is a path from the point immediately following $d$ to $p$, such that nodes in the path do not redefine $v$. Reaching definitions at a block $B$, RD($B$), is the set of definitions that reach the entry point of $B$ from the idle node. The algorithm to compute reaching definitions at a block in an ICFG is similar to that to compute reaching definitions at a node in a CFG. The later algorithm can be found in [1] and therefore we omit our algorithm here due to space limit. The time complexity of this algorithm (Table 2) is $O(|V_S^C|^2|N_C|)$, where $|V_S^C|$ is the number of shared variables in $C$, and $|N_C|$ is the number of blocks in $G_{ICFG}(C)$. We use DU($C$) to represent the set of all def-use associations in a component $C$, including those computed by `getDUs` (Table 2). The following criterion is a natural extension of all-uses.

CRITERION 2. *A test suite* T *satisfies the* **all-inter-context-uses** *adequacy criterion for a nesC application if and only if for each def-use association* dua *in DU(C), at least one test case in* T *covers* dua.

The all-inter-context-uses criterion subsumes the all-uses criterion: all-inter-context-uses not only requires test cases to cover def-use associations in individual flow graphs, but also to cover those of shared variables in components. Intuitively, the all-inter-context-uses criterion is useful to explore different sequences of events related to shared memories in the concurrent calling context setting. Note that other control-flow and data-flow criteria, such as all-preemption-edges and all-inter-context-du-path, can be similarly developed based on ICFGs, but we do not present them in this paper due to page limit.

## 5. EXPERIMENT

We conducted experiments to study whether test suites satisfying our proposed test adequacy criteria are effective in exposing faults in nesC applications. We compared all-branches [16] (all-uses [9]) with its extended version all-inter-context-edges (all-inter-context-uses). We made the comparison to answer the following two questions.

Q1: Are test suites that satisfy all-inter-context-edges (all-inter-context-uses) criterion more effective than test suites that satisfy all-branches (all-uses) criterion in exposing faults?

Q2: If the answer to Q1 is yes, is all-inter-context-edges (all-inter-context-uses) criterion more effective than all-branches (all-uses) criterion simply because all-inter-context-edges (all-inter-context-uses) requires more test cases?

## 5.1 Experimental Setup

### 5.1.1 *Subject Application*

We used Sentri [19], a structural health monitoring WSN application, in our study. Structural health monitoring is a technology which estimates structural states and detects structural changes that impact how a structure performs. The studied application is designed and deployed on the 4200ft long main span and the south tower of the Golden Gate Bridge (GGB). Following the metrics adopted by Cooprider et al. [7] and Henzinger et al. [13], Sentri contains 22681 lines of code, 71 components, and 50 interfaces. Excluding the code of the underlying operating systems, Sentri contains 22 components and 14 interfaces. The figures of test coverage information in Section 5.3 were based on these 22 components.

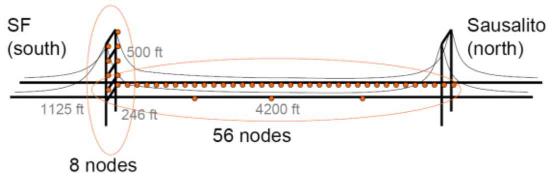


**Figure 5. Deployment of Sentri on the GGB [19].**

### 5.1.2 *Seeding Faults*

We opted to seed faults into the source code of Sentri, because there are no enough identifiable faults in the CVS repository, and mutation operators for nesC programs are unavailable. We conducted fault analysis and created fault classes that are used as "templates" to create instances of similar faults [32]. We extracted 16 fault classes from two fault analysis processes, including (1) analysis of bug reports and a CVS repository of TinyOS[3], and (2) analysis of nesC language constructs that support the programming model of TinyOS. Based on these fault classes, we seeded 64 faults into the source code, producing 64 faulty versions with one fault in each version. Each fault instance was instantiated randomly from the 16 fault classes, and was seeded at an arbitrary location in the source code in such a way that the resultant faulty version can be successfully compiled. This fault-seeding process bears an analogy with that used by Andrews et al. [3]. We ran the 64 faulty versions over the entire pool of test cases, referred to as *test pool*. We explain how to generate the test pool in Section 5.2. We excluded the results of seven faulty versions that have failure rates above 0.1 from our data analysis. A faulty version's *failure rate* is the proportion of failed test cases in the test pool. We also excluded the results of eight faulty versions that no test case in the test pool can fail them. Our approach to exclude the results of faulty versions follows the practice of existing work [17][22]. The remaining 49 faulty versions' failure rates range from 0.001 to 0.095, with a mean of 0.024. We conducted data analysis based on the results of these 49 versions.

### 5.1.3 *Simulation Setup*

We conducted our experiments using a sensor network simulator Avrora [29]. We developed our testing framework based on the testing frameworks of interrupt-driven software [26] and GUI applications [32]. We used Java and Python to implement our framework. A test input in our framework contains three parts: simulation configuration, message sequences, and interrupt schedules. A test case includes a test input and the corresponding correctness verdict. We developed a driver mote to send messages to the mote under test to facilitate automated testing.

**Simulation configuration**. We carried out our experiments on Avrora with a simulated Mica2 platform and ATMega128 microcontroller, and provided sensors with random values. The simulation time for each test case execution was chosen arbitrarily between 1ms and 20000ms. We chose an upper bound of 20000ms so we can complete our experiments in a reasonable time. The simulation time decides the number of messages sent to the mote under test and the number of interrupt occurrences in a test case. For each test input, we randomly chose a simulation time to factor out its impact on the result. The average simulation time in our experiments is 10449ms.

**Message sequences**. The driver mote randomly sends messages (packets) to the test mote using the `SendMsg` interface provided by TinyOS. The interval between two successive messages was arbitrarily sampled from 1ms to 1000ms. We selected 40 random test suites for each test-suite size (1-199). The average number of messages sent by the driver mote in this setting is 25.02.

**Interrupt schedules**. An interrupt schedule is a list of 2-tuple (*vector*, *time*) ordered by ascending *time*, where *vector* is the interrupt number and *time* decides when to fire the interrupt. It is important to choose a proper interrupt schedule. If interrupts occur rarely, there would be no preemption of tasks. On the other hand, if interrupts occur too often, interrupt handlings would occupy the corresponding mote. The interrupt schedule was so chosen that the mote under test spent fifty percent of its cycles on handling interrupts, as suggested by Regehr [26].

Avrora runs a simulation when it has received a test input and stops when the specified simulation time is over. Our tool collected the outputs of each simulation, and compared them with those given by the "golden" version, which was used as oracle information [32]. If our tool did not find any deviation in the comparison, it marked the corresponding test case of that simulation as passed. Outputs collected from each simulation include: (1) the state transitions of LED, (2) the messages sent from a radio, (3) the messages sent from UART, and (4) the changes of I/O registers in EEPROM.

## 5.2 Experimental Design

We designed our experiments to study the fault-detection effectiveness of the two proposed test adequacy criteria. We measured the effectiveness by the fault-detection rate [8] as follows.

$$af(F_i, T_s) = \frac{df(F_i, T_s)}{|T_s|}, \quad af(F, T_s) = \frac{1}{|F|}\sum_{F_i \in F} af(F_i, T_s)$$

$T_s$ is a set of test suites, where each $T \in T_s$ either satisfies some criteria or has a fixed size, and $|T_s|$ is the number of test suites in $T_s$. $Df(Fi, T_s)$ is the number of test suites in $T_s$ that expose fault $F_i$, where $i$ belongs to the indices of faulty versions. A test suite $T$ exposes a fault $F_i$ if any test case $t$ in $T$ exposes $F_i$. $Af(F_i, T_s)$ is the fault-detection rate of $T_s$ with respect to $F_i$ and $af(F, T_s)$ is the average fault-detection rate of $T_s$ with respect to a set of faults $F$.

We produced 2000 test cases using the approach described in Section 5.1.3 to form a test pool. We limited the number of test cases to 2000 so we could complete the experiment in a reasonable time. Our experiment covered simulations of 64 faulty versions over the entire test pool.

To answer the first research question Q1, we selected test suites that satisfy various criteria. We added a test case to a test suite only when the test case increased the coverage of the current test suite. The selection stopped when the test suite satisfied 100% coverage of a criterion or reached an upper bound. The upper bounds were randomly chosen from [1, $M$], where $M$ is a number slightly larger than the size of the largest test suites reaching 100% coverage, over all criteria. Other researchers [17][22] also used

[3] http://sourceforge.net/projects/tinyos/

this test selection approach for their studies. We considered uncovered edges and def-use associations as infeasible and excluded them from computing the coverage. We selected forty test suites for each combination of versions and upper bounds.

To answer Q2, we selected 7960 (= 40*199) test suites so there were 40 test suites for each size ranging from 1 to 199. There were a few situations where we did not have 40 test suites of a specified size. We used two approaches to enhance a smaller test suite with additional test cases until it reached the specified size. We call test suites thus formed *random-enhanced* and *repetition-enhanced*, respectively. A random-enhanced test suite was enhanced with additional randomly sampled test cases. A repetition-enhanced test suite was filled with additional test cases by resetting its test coverage and repeating the same test case selection process. We used both types of test suites to address two internal threats to validity of our analysis. If a criterion, $C$, requires far fewer numbers of test cases than the specified size, random-enhanced test suites of criterion $C$ can be more or less like test suites selected by random. On the other hand, if the test pool does not have enough test cases to satisfy criterion $C$ repetitively in one test suite, the repetition-enhanced test suites of $C$ will not reach the specified size. We used both types of test suites simultaneously to alleviate these threats. We also used the random criterion as a benchmark.

## 5.3 Data Analysis

In this section, we present the results of our analysis, addressing each research question in turn.

### 5.3.1 *Effectiveness of Adequacy Criteria*

**Table 3. Overall fault-detection rates and coverage.**

| Criterion | Fault-Detection Rate | | | Coverage | | |
|---|---|---|---|---|---|---|
| | Min | Mean | Max | Min | Mean | Max |
| all-branches | 0.00 | 0.73 | 1.00 | 0.85 | 0.96 | 1.00 |
| all-inter-context-edges | 0.03 | 0.89 | 1.00 | 0.71 | 0.83 | 0.98 |
| all-uses | 0.00 | 0.55 | 0.97 | 1.00 | 1.00 | 1.00 |
| all-inter-context-uses | 0.01 | 0.93 | 1.00 | 0.87 | 0.95 | 1.00 |

We first compared the criteria irrespective of the failure rates of seeded faults. The average fault-detection rates, $af(F, T_s)$, of selected test suites and coverage of criteria are given in Table 3. The statistics were computed over all test suites produced to answer Q1. On average, all-inter-context-uses has the highest fault-detection rate, followed by all-inter-context-edges. We observed from the experiment that all-inter-context-edges is 21% more effective than all-branches. Similarly, all-inter-context-uses is 69% more effective than all-uses. Although all-uses is effective to test conventional programs, our results suggest that all-uses is less effective in exposing faults than all-branches. A likely reason is that a non-redundant test suite with respect to all-uses may cover less code than that with respect to all-branches in the event-driven programming paradigm. For instance, any test suite trivially satisfies all-uses for a program that contains only one event handler $E$, where $E$ defines only shared variables (and without any variable uses). On the other hand, only test suites with at least one test case invoking $E$ for this code can satisfy all-branches.

To study the average fault-detection rate of each criterion with respect to different faulty versions, we used the k-means clustering algorithm [18] to partition the faults into three categories. We label these three categories as hard faults, medium faults, and easy faults, according to the relative average failure rates of the faulty versions in each category. The result is shown in Figure 6. From Figure 6, we observed that all-uses is always less effective than all-inter-context-uses in all categories, and particularly in the hard-fault category. Intuitively, without considering data flows caused by interrupts and tasks, the average fault-detection rate of all-uses in the hard-fault category is only one third of that of all-inter-context-uses in the same category. A similar observation exists between all-branches and all-inter-context-edges in the hard- and medium-fault categories although the difference is less significant.

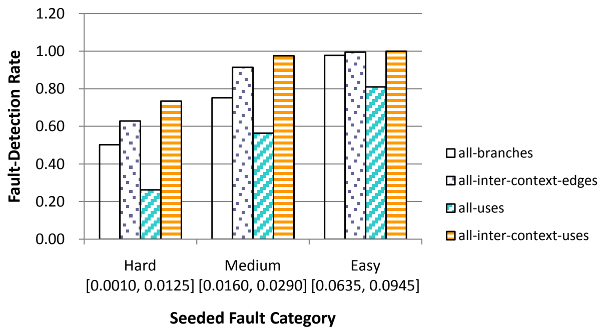


**Figure 6. Fault-detection rates of different fault categories.**

We inspected the code of the faulty versions in the hard-fault category. We found that they are faults more specific to nesC applications, such as interface-contract violation and data races. For closer examination, we compared the effectiveness of different criteria in exposing these two kinds of faults across the 49 faulty versions. Figure 7 gives the average fault-detection rates of all five such faults. Four of them remain in the hard-fault category. We observed from Figure 7 that all-inter-context-edges has the best performance, followed by all-inter-context-uses.

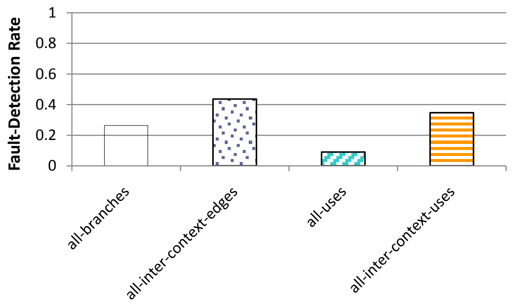


**Figure 7. Fault-detection rates over synchronization faults.**

### 5.3.2 *Cost-Effectiveness of Adequacy Criteria*

It is common that different criteria need different sizes of test suites. In this section, we study whether the increased effectiveness of all-inter-context-edges (all-inter-context-uses) beyond that of all-branches (all-uses) is attributed to increased test-suite sizes or other factors of these criteria. To answer this question, we used a regression model to fit the average fault-detection rates over various test-suite sizes (1-199) and studied the cost-effectiveness of different criteria based on the model. A regression model can capture the intrinsic relation between test-suite sizes and fault-detection rates [3]. The model we used is $af = 1 - a^{b \cdot size}$ $(0 < a < 1, b > 0)$, where $af$ and $size$ stand for $af(F, T_s)$ and test-suite size, respectively. This model presents three properties. (1) When there are no test cases ($size = 0$), the fault-detection rate is trivially zero. (2) When a test suite includes infinitely many test cases, the fault-detection rate approximates one hundred percent. (3) When the test-suite size grows, the fault-detection rate increases more slowly. We used least square fitting [15] to estimate the parameters. In other words, we found the values of $a$ and $b$ to minimize $E^2(a, b)$:

$$E^2(a, b) = \sum_{j=1}^{N} (af_j - (1 - a^{b \cdot size_j}))^2$$

$N$ is the number of data points (199), $size_j$ equals $j$, and $af_j$ stands for $af(F, T_s^j)$, where $T_s^j$ is a set of test suites with size $j$. Although it is difficult to get a closed-form expression to this problem, we can use numerical methods to get an approximate solution, such as the downhill simplex method.

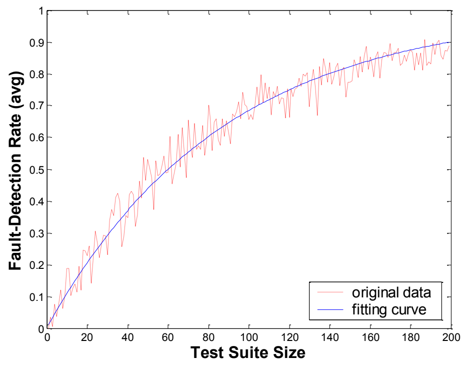


**Figure 8. Original data and curve fitting for random.**

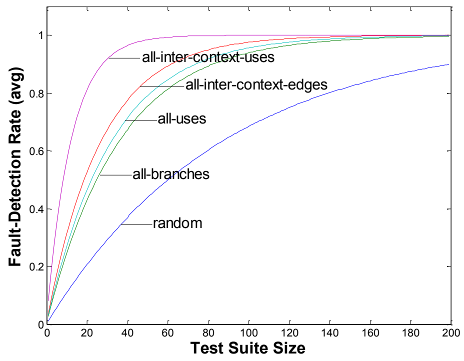


**Figure 9. Curve fitting for random-enhanced test suites.**

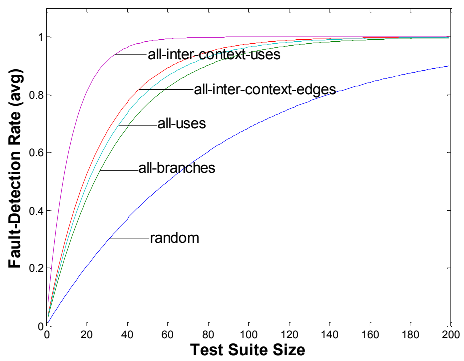


**Figure 10. Curve fitting for repetition-enhanced test suites.**

Figure 8 shows the original data and the fitted curve for random criterion. Figure 9 and Figure 10 show the fitted curves of different criteria for random-enhanced and repetition-enhanced test suits, respectively. Table 4 gives the estimated parameters and least square errors. Although the original data of the same criterion have different shapes in random-enhanced and repetition-enhanced test suites, it is interesting that the estimated parameters of the same criterion for these two types have similar values. In addition, the relative positions of curves for any two criteria do not change between Figure 9 and Figure 10.

**Table 4. Estimated parameters.**

| | | random | all-branches | all-inter-context-edges | all-uses | all-inter-context-uses |
|---|---|---|---|---|---|---|
| random-enhanced | $a$ | 0.987 | 0.971 | 0.961 | 0.968 | 0.922 |
| | $b$ | 0.896 | 0.959 | 0.949 | 0.948 | 1.027 |
| | $E^2(a, b)$ | 0.383 | 0.348 | 0.314 | 0.322 | 0.328 |
| repetition-enhanced | $a$ | 0.987 | 0.969 | 0.962 | 0.965 | 0.921 |
| | $b$ | 0.896 | 0.944 | 0.949 | 0.938 | 1.014 |
| | $E^2(a, b)$ | 0.383 | 0.429 | 0.331 | 0.375 | 0.361 |

**Table 5. Adequacy criteria areas normalized by random.**

| | all-branches | all-inter-context-edges | all-uses | all-inter-context-uses |
|---|---|---|---|---|
| random-enhanced | 1.348 | 1.421 | 1.379 | 1.542 |
| repetition-enhanced | 1.358 | 1.420 | 1.394 | 1.541 |

**Table 6. Predicted test-suite sizes (max fault-detection rates).**

| | | | |
|---|---|---|---|
| random-enhanced | $\min\{\max af_i^{\text{all-branches}}, \max af_i^{\text{all-inter-context-edges}}\} = 0.939$ | | |
| | predicted size | | ratio |
| | all-branches: 107 | all-inter-context-edges: 82 | 1.305 |
| | $\min\{\max af_i^{\text{all-uses}}, \max af_i^{\text{all-inter-context-uses}}\} = 0.965$ | | |
| | predicted size | | ratio |
| | all-uses: 115 | all-inter-context-uses: 48 | 2.396 |
| repetition-enhanced | $\min\{\max af_i^{\text{all-branches}}, \max af_i^{\text{all-inter-context-edges}}\} = 0.925$ | | |
| | predicted size | | ratio |
| | all-branches: 97 | all-inter-context-edges: 77 | 1.260 |
| | $\min\{\max af_i^{\text{all-uses}}, \max af_i^{\text{all-inter-context-uses}}\} = 0.860$ | | |
| | predicted size | | ratio |
| | all-uses: 67 | all-inter-context-uses: 31 | 2.161 |

There are several aspects that we can compare adequacy criteria based on the fitted curves.

1. At first glance, the fitted curves of all-inter-context-uses rise faster than those of all-uses, which implies that all-inter-context-uses is more effective than all-uses over all the sizes. The relation between all-inter-context-edges and all-branches is similar, though the difference is less significant.
2. One way to quantitatively compare cost-effectiveness of test adequacy criteria is to compare the area under these fitted curves [3]. Intuitively, the larger is the area, the better is the criterion. The adequacy-criteria areas normalized by the random-criterion area are shown in Table 5. We observed from the table that all-inter-context-edges is better than all-branches, and all-inter-context-uses is better than all-uses.
3. Table 6 shows the predicted test-suite size of individual criteria at their maximum fault-detection rate in the original data. This is another interesting metrics to compare. For example, the first three rows show that the maximum fault-detection rate of all-branches is 0.939 using random-enhanced test suites. In the fitting model, all-branches needs at least 107 test cases to reach this fault-detection rate, while all-inter-context-edges needs 82 test cases to reach the same rate. In other words, all-branches needs 30.5% more test cases than all-inter-context-edges to reach its maximum fault-detection rate of 0.939. This percentage is similar to the result (26.0%) using repetition-enhanced test suites. Similar observation is made between the results of all-uses and all-inter-context-uses.

## 5.4 Threats to Validity

In this section, we discuss threats to internal and external validities of our experiments, and the measures we took to address the threats.

Our testing framework involves many tools. Any of them could have added variability to our result, thus threatening internal validity. Two major processes in our experiment are test case generation and test pool construction. The test pool included 2000 randomly produced test cases. A threat could occur when selecting repetition-enhanced test suites in answering Q2, because there may not be enough test cases in the pool to cover a certain criterion repetitively in one test suite. We used random-enhanced test suites to compensate this demerit. The experimental results between these two types of test suites are consistent. We have discussed the threat arising from test case generation in Section 5.1.3.

Several issues may affect generalizing our results, thus threatening external validity. The first issue is whether the programming model used is representative. Interrupt handling and task queuing is not specific to nesC applications running on top of TinyOS [14]. They are also common in other networked embedded systems written by other languages running on top of operating systems, such as TinyGALS [6], and SOS [12]. The second issue is whether the subject application and the seeded fault classes are representative. Sentri contains about 22 KLOC, which is a real-life WSN application that has been deployed on the Golden Gate Bridge. We used two fault analysis processes to extract fault classes: (1) analysis of TinyOS CVS repository and bug report, and (2) analysis of nesC language constructs that support TinyOS's programming model. The third issue is whether our testing processes are representative. Testing networked embedded systems based on simulations is a popular means to assure the quality of these systems [21][29]. We chose the simulator Avrora [29] that has high fidelity to the real world. Avrora supports interrupt preemptions, heterogeneous network communication, and resource-constraint checking, which are the three major concerns in developing networked embedded systems. Finally, we did not compare some adequacy criteria devised for concurrent programs, such as [33], because subroutines in nesC applications are not related by synchronization primitives. These techniques also do not capture the task posting semantics in nesC. Our approach leverages scheduling heuristics in TinyOS to reduce infeasible interleaving in nesC applications. Therefore, our approach saves efforts in analyzing source code and generating tests compared with these concurrent testing techniques.

## 6. RELATED WORK

Conventional testing techniques were proposed mainly for programs with non-preemptive execution, such as control-flow testing criteria [16] and data-flow testing criteria [9]. Researchers extended these criteria to programs with more than one subroutine, where inter-procedural flow information [11] is used. These inter-procedural testing methods represent programs by relating flow graphs of individual subroutines via subroutine calls. Data-flow testing approaches are also extended for concurrent programs [20][33] where thread interactions are restricted to synchronization codified in the programs. Our test adequacy criteria are different from all of them because interrupt handling subroutines and tasks are related by neither subroutine calls nor synchronization primitives.

There are several pieces of work focusing on producing a representation of an application of similar kinds. Unlike Brylow et al. [5] who proposed to represent interrupts in the assembly language, we model interrupt preemptions at the application layer , and we also model task postings. Memon et al. [24] showed event interactions for GUI applications using event-flow graphs, which do not capture the notion of preemptions. Nguyen et al. [25] proposed application posting graphs to represent behaviors of WSN applications. Unlike our approach, this representation does not capture resumptions of tasks after they are preempted. However, it is likely that a shared variable is defined in an interrupt handler preempting a task, and subsequently used by the task after the task resumes its execution. This def-use association provides an opportunity to expose faults as shown in Section 3.

Our work falls in the scope of testing ubiquitous systems. Regehr [26] designed a restricted interrupt discipline to enable random testing of nesC applications. Superior fault detection effectiveness of our proposed criteria in comparison with random criteria was empirically evaluated in Section 5.3.2. Metamorphic testing was explored to alleviate the test oracle problem encountered in testing context-aware middleware centric applications [30]. In that paper, Tse et al. pinpointed that conventional data-flow testing techniques are inadequate for ubiquitous software. Lu et al. [22][23] studied data-flow based adequacy criteria for testing context-aware applications. We study inter-context control-flow and data-flow adequacy criteria in nesC applications where preemption and resumption of program subroutines have not been studied. Wang et al. [31] described an approach to generate tests for context-aware applications by manipulating context data. Their approach mainly targets at exposing faults due to interleaving of event handlers in context-aware applications. Their approach, however, does not account for deferred execution of tasks in nesC applications. We note that the notion of context in this paper refers to a sequence of subroutine calls; whereas in ubiquitous computing, a context usually refers to the interesting information of the computing environment of computing entities.

Simulation is often adopted to study system design and validate system correctness for distributed systems. Testing based on simulation is a way in verifying the correctness of networked embedded systems [21][29]. Rutherford et al. [27] explored the idea of using simulation as a basis for defining test adequacy criteria. They treated the simulation code as a specification to test the implementation. Our approach directly runs and tests the implementation on the simulator Avrora.

## 7. CONCLUSION

NesC is a commonly used language for programming networked embedded systems that are emerging. The high degree of interrupt-based concurrency makes programming nesC applications an error-prone task. A faulty application may mishandle a shared storage and produce the wrong results if its developer has not carefully programmed the non-blocking execution of interrupts and deferred execution of tasks. To test these systems systematically, we have introduced inter-context flow graphs to express behaviors of nesC applications and proposed two test adequacy criteria for these systems. Our criteria focus on the coverage of interrupt preemptions that can affect application behaviors. We conducted simulation experiments to evaluate our proposed criteria. Empirical results show that our approach is promising in exposing faults in nesC applications. In the future, we plan to perform more empirical studies to evaluate our proposal. In particular, we would like to study the preciseness of our def-use association analysis when nesC applications grow larger, and how this preciseness affects the effectiveness of our proposed test adequacy criteria.

## 8. ACKNOWLEDGMENT

We would like to thank Tao Xie and Hong Zhu for fruitful discussions in developing the ideas for this paper. We also thank the anonymous reviewers for their constructive comments on an early draft of this paper.

## 9. REFERENCES

[1] Aho, A. V., Sethi, R., and Ullman, J. D. 1988. *Compilers: Principles, Techniques, and Tools*, Chapter 10. Addison-Wesley Pub. Co., 1988.

[2] Ammons, G., Ball, T., and Larus, J. R. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.* 32, 5 (May. 1997), 85-96.

[3] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.* 32, 8 (Aug. 2006), 608-624.

[4] Archer, W., Levis, P., and Regehr, J. 2007. Interface contracts for TinyOS. In *Proc. of IPSN* '07, 158-165.

[5] Brylow, D., Damgaard, N., and Palsberg, J. 2001. Static checking of interrupt-driven software. In *Proc. of ICSE* '01, 47-56.

[6] Cheong, E., Liebman, J., Liu, J., and Zhao, F. 2003. TinyGALS: a programming model for event-driven embedded systems. In *Proc. of SAC* '03, 698-704.

[7] Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. 2007. Efficient memory safety for TinyOS. In *Proc. of SenSys* '07, 205-218.

[8] Frankl, P. G. and Weiss, S. N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.* 19, 8 (Aug. 1993), 774-787.

[9] Frankl, P. G. and Weyuker, E. J. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.* 14, 10 (Oct. 1988), 1483-1498.

[10] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The *nesC* language: A holistic approach to networked embedded systems. In *Proc. of PLDI* '03, 1-11.

[11] Harrold, M. J. and Soffa, M. 1994. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (Mar. 1994), 175-204.

[12] Han, C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. SOS: A dynamic operating system for sensor networks. In *Proc. of MobiSys* '05.

[13] Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Race checking by context inference. In *Proc. of PLDI* '04, 1-13.

[14] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.* 34, 5 (Dec. 2000), 93-104.

[15] Hill, T. and Lewicki, P. 2007. *STATISTICS Methods and Applications*. StatSoft, Tulsa, OK, 2007.

[16] Huang, J. C. 1975. An approach to program testing. *ACM Comput. Surv.* 7, 3 (Sep. 1975), 113-128.

[17] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE* '94, 191-200.

[18] Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. 2002. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 7 (Jul. 2002), 881-892.

[19] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. 2007. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. of IPSN* '07, 254-263.

[20] Lei, Y., and Carver, R. H. 2006. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.* 32, 6 (Jun. 2006), 382-403.

[21] Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of SenSys* '03, 126-137.

[22] Lu, H., Chan, W. K., and Tse, T. H. 2006. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proc. of SIGSOFT* '06/*FSE*-14, 242-252.

[23] Lu, H., Chan, W. K., and Tse, T. H. 2008. Testing pervasive software in the presence of context inconsistency resolution services. In *Proc. of ICSE* '08, 61-70.

[24] Memon, A. M., Soffa, M. L., and Pollack, M. E. 2001. Coverage criteria for GUI testing. In *Proc. of ESEC/FSE*-9, 256-267.

[25] Nguyen, N. T. and Soffa, M. L. 2007. Program representations for testing wireless sensor network applications. In *Proc. of DOSTA* '07, 20-26.

[26] Regehr, J. 2005. Random testing of interrupt-driven software. In *Proc. of EMSOFT* '05, 290-298.

[27] Rutherford, M. J., Carzaniga, A., and Wolf, A. L. 2006. Simulation-based test adequacy criteria for distributed systems. In *Proc. of SIGSOFT* '06/*FSE*-14, 231-241.

[28] TinyOS Tutorials. *Modules and the TinyOS execution model*. http://docs.tinyos.net/index.php/Modules_and_the_TinyOS_Execution_Model.

[29] Titzer, B. L., Lee, D. K., and Palsberg, J. 2005. Avrora: scalable sensor network simulation with precise timing. In *Proc. of IPSN* '05, 477-482.

[30] Tse, T. H., Yau, S. S., Chan, W. K., Lu, H., and Chen, T. Y. 2004. Testing context-sensitive middleware-based software applications. In *Compsac* '04, 458-466.

[31] Wang, Z., Elbaum, S., and Rosenblum, D. S. 2007. Automated generation of context-aware tests. In *Proc. of ICSE* '07, 406-415.

[32] Xie, Q. and Memon, A. M. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.* 16, 1 (Feb. 2007), 4.

[33] Yang, C. D., Souter, A. L., and Pollock, L. L. 1998. All-du-path coverage for parallel programs. In *Proc. of ISSTA* '98, 153-162.

[34] Zhang, Y. and West, R. 2006. Process-aware interrupt scheduling and accounting. In *Proc. of RTSS* '06, 191-201.

[35] Zhu, H., Hall, P. A., and May, J. H. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366-427.